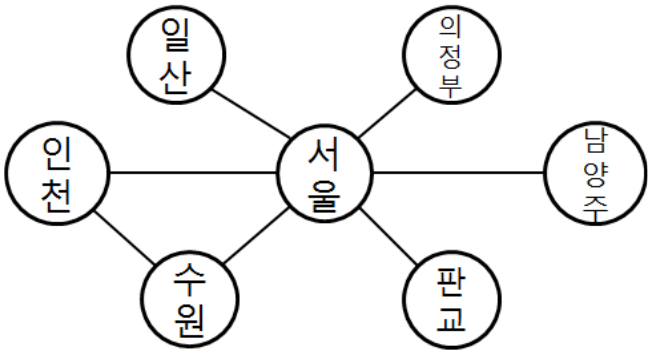
<알고리즘 Mix Tape 04>

8장_1 그래프 입문하기 (170529 to 170531)

그래프를 생각해보면 우리가 처음에 도표를 이용해 수량 혹은 변화를 그린 것을 살펴볼 수 있고, x축과 y축으로 갖고 노는 평면 그래프를 생각해 볼 수 있다. 하지만 우리가 알고리즘에서 공부하는 그래프는 이산수학에서 정점과 간선으로 갖고 노는 그 개념이다. 이번 시간에는 그래프에 대해서 간단히 복습하면서, BFS(너비 우선 탐색), DFS(깊이 우선 탐색)에 대해서 자세히 배워보는 시간을 가져보자.

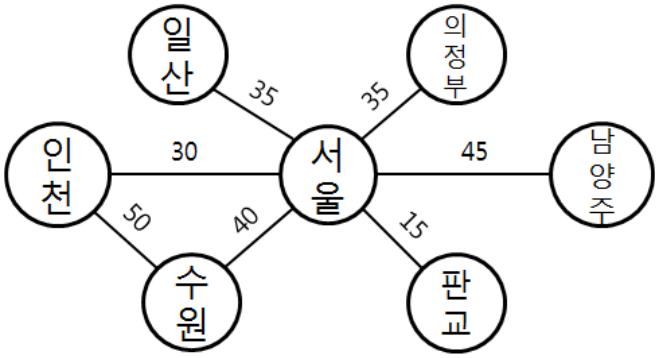
1. 그래프(Graph)

우리가 이산수학, 자료구조 시간에 잠깐 그래프에 대해 배운 적이 있다. 그래프 이론이 마치 사골곰탕 우려먹듯이 나오는 이유가 그만큼 매우 중요하다는 뜻으로 받아들여야 할 필요가 있다. 여기서 그래프의 간단한 사례를 살펴보도록 하자.

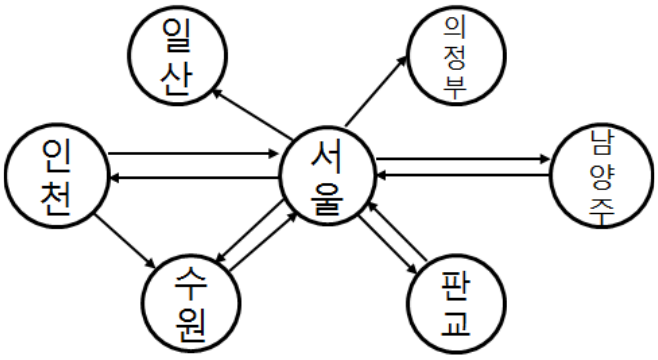


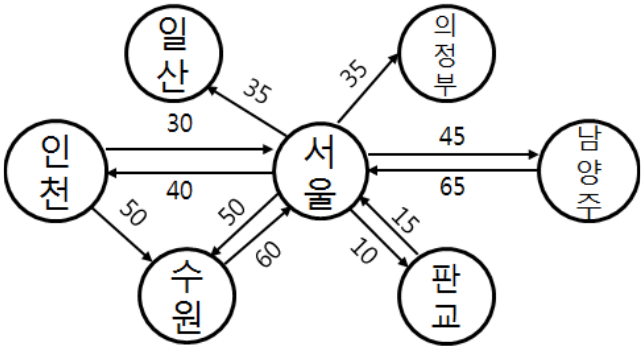
여기서 그린 그래프의 개념은 각 도시 별로 이어주는 도로를 통해 갈 수 있는 경로에 대해 그래프로 표현을 해봤다. 예를 들어 서울을 기점으로 치면 인천은 경인고속국도를 이용할 수 있고, 남양주까지는 경춘로를 이용할 수 있고, 판교, 수원은 경부 고속국도를 이용할 수 있고, 의정부는 경원로를 이용할 수 있고, 일산은 통일로, 자유로를 이용해서 갈 수 있다. 이처럼 도시 간 이어주는 도로를 우리는 "간선(Edge)"로 생각하면 쉽고, 각 도시(서울, 수원, 인천 etc...)들은 "정점(Vertex)"으로 생각하면 그래프에 대해서 쉽게 접근할 수 있다.

그래프의 자료 구조는 노드로 표현을 한다면 대개 정점(Vertex)이 핵심 요소가 되고, 각 정점별로 이어주는 선이 바로 간선들(Edge)로 정리할 수 있다. 그리고 서울에서 모든 도시들이 접근이 가능하니깐 경부, 경인 고속국도, 자유로, 통일로, 경춘로 등등을 통해서 접근을 하게 하니 한 정점에서 간선으로 연결된 경우를 인접(Adjacent)으로 살펴보면 되겠다. 이제 그래프에 숫자를 한번 첨가해 보도록 하자.



각 간선에 숫자가 있는 모습을 볼 수 있다. 이를 가중치(Weighted Value)라고 칭한다. 여기서 가중치는 걸리는 시간, 걸리는 길이, 혼잡도 등등으로 따져볼 수 있는데 우리는 걸리는 시간으로 잡고 생각해보고 넘어가겠다. 그런데 아이러니한 것은 도로에는 분명 상행선이 있고, 하행선이 있다. 어쩌면 상행선은 혼잡해서 실제 시간보다 오래 걸릴 수 있고 혹은 하행선은 검문 중으로 이용 제한이 있는 경우도 살펴볼 수 있다. 이번에는 아까 그래프에서 화살표로 바꿔보도록 하자.

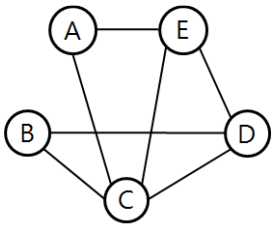


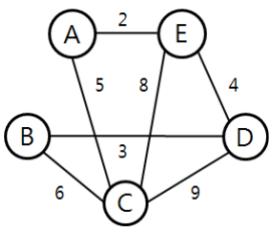


이처럼 화살표로 나타낸 그래프를 유향(有向)그래프라고 한다. 물론 위처럼 가중치를 안 쓴 경우도 있고, 아래처럼 가중치를 적은 경우도 있다. 여기서 서울에서 수원과의 인접 관계를 살펴보면 서울에서 수원으로 내려가는 경우에는 시간이 50 정도 걸리는데 수원에서 서울로 올라가는 경우는 교통 체증으로 인해 60이 걸리게 된다. 아니면 인천에서 수원으로 내려가는 경우에는 50 정도 걸리는데 수원에서 인천으로 올라가는 상행선이 검문으로 인하여 막히거나 도로 보수 중인 경우에는 못 올라가게 되어 서울을 경유해 인천으로 가는 방법 이외엔 없다. 이처럼 그래프를 표현하는 방법으로서는 여러 가지 방법이 존재한다.

2. 그래프 자료구조

그래프는 어떻게 표현이 될까? 솔직히 정점에 서울, 수원, 인천까지 쓰긴 뭐하니깐 알파벳을 통해서 새로 그래프를 만들었으니 이를 통해서 작성을 해보도록 하자.





<그래프1(左)> 가중치 없는 일반 그래프
<그래프2(右)> 가중치 있는 일반 그래프

2-1) 그래프 1를 행렬로 표현하자

우리가 이산수학 시간에 그래프를 행렬로 표현한 경우가 있을 것이다. 그래프 1를 행렬로 표현을 해보면 아래와 같이 작성할 수 있다. 일단 그래프의 정점의 개수를 파악하면 5개이다.

int[][] graph=new int[5][5];

행렬로 표현하려면 정점의 개수의 제곱만큼 표현을 해줘야 경로에 따라서 표현하는데 무리가 없다. 일단 (v1, v2)에 경로가 있다면 (v2, v1)도 경로가 있다는 점을 유의하면 쉽게 작성할 수 있다.

	A	B	C	D	E
A	0	0	1	0	1
B	0	0	1	1	0
C	1	1	0	1	1
D	0	1	1	0	1
E	1	0	1	1	0

	A	B	C	D	E
A	0	0	5	0	2
B	0	0	6	3	0
C	5	6	0	9	8
D	0	3	9	0	4
E	2	0	8	4	0

여기서 검은색으로 칠한 것은 무엇일까? 윤동주의 자화상에서는 한 점도 부끄럼이 없다는 뜻으로 거울을 통해 자화상을 살펴보듯이, 그래프에서 거울과 같은 역할로 한 정점도 빠짐없이 연결됨을 확인하기 위한 개념으로 생각하면 쉽게 행렬로 표현이 가능하다.

또 하나 재미있는 점은 왼쪽에 있는 행렬에서 (행렬의 요소의 합)/2를 하면 그래프의 간선의 수가 나온다는 점이다. 그리고 한 행의 요소의 합(혹은 한 행의 요소의 합)은 한 정점의 차수(한 정점에서 다른 정점으로 연결된 간선의 수)가 나온다.

그렇지만 실제로 그래프 이론을 접근하는 경우에 행렬로 접근하는 것은 시간 복잡도에 악영향을 미쳐서 많이 안 쓰는 편이다. 왜냐면 정점이 부지기수한데 정작 연결된 간선은 다각형인 경우에 희소 행렬(0만 오지게 많은 행렬)이 나오기 때문에 비효율적이기 때문이다.

그럼 이를 대처하기 위해서는 또 다른 표현 방법이 있는데 리스트로 표현하는 방법을 소개하겠다.

2-2) 그래프 1를 리스트로 표현해보자

A -> C -> E
B -> C -> D
C -> A -> B -> D -> E
D -> B -> C -> E
E -> A -> C -> D

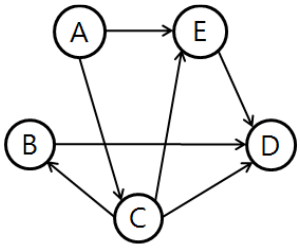
A -> C 5 -> E 2
B -> C 6 -> D 3
C -> A 5 -> B 6 -> D 9 -> E 8
D -> B 3 -> C 9 -> E 4
E -> A 2 -> C 8 -> D 4

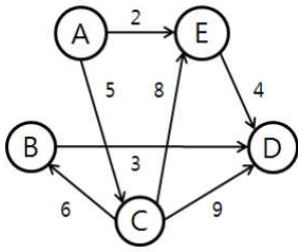
처음 강의 노트에 있는 걸 표현하려면 비지오로 편집하기 귀찮아서 표로 만들었다. 일단 정점 노드를 통해서 옆에다가 서로 인접하는 정점들에 대한 리스트를 작성해줘서 이용하는 방법을 대개 많이 이용하고 있다. 우리가 리스트로 표현을 하는데 있어서 대개 **오름차순으로 작성을 하는 점을 주목**해야 한다. 이는 앞으로 배울 **BFS, DFS(각각 너비, 깊이로 우선 탐색)에서 많은 영향을 끼치기 때문**에 이로 표현하는 것을 꼭 알고 넘어갔으면 좋겠다.

물론 그래프에는 가중치가 들어가기 마련이다. 가중치가 들어가기 마련이다. 2-1)처럼 그래프를 표현한 행렬을 위상 행렬이라고 칭하는데 위상 행렬에 1 대신에 서로 인접할 때 쓰는 가중치를 작성을 하면 된다. 뭐 인접이 안 되었다면 0으로 쓰면 되겠고...

또한 2-2)처럼 인접 리스트에서는 음영으로 칠해진 부분에 가중치라는 변수를 리스트로 입력받아서 아래처럼 작성하면 되겠다.

(참고) 유향 그래프는 아래와 같이 작성된다고 볼 수 있다.





<그래프3(左)> 가중치 없는 유향 그래프

<그래프4(右)> 가중치 있는 유향 그래프

2-1)처럼 행렬로 표현을 하면 아까 이야기한 윤동주의 자화상 이야기는 뒤로 하고 화살표로 이어진 인접에 대해서만 1를 넣든 가중치를 넣든 하면 된다. 여기서 유의해야 하는 부분은 시발점이 행을 기준으로 하고, 종착점을 열을 기준으로 해야 한다.

참고로 그래프3의 위상 행렬(인접 행렬)의 요소의 합은 여기서는 인접된 정점 연결 수의 합이다. 각 행의 요소의 합은 진출 차수이고, 각 열의 요소의 합은 진입 차수의 합이다.

	A	B	C	D	E
A	0	0	1	0	1
B	0	0	0	1	0
C	0	1	0	1	1
D	0	0	0	0	0
E	0	0	0	1	0

	A	B	C	D	E
A	0	0	5	0	2
B	0	0	0	3	0
C	0	6	0	9	8
D	0	0	0	0	0
E	0	0	0	4	0

이제 2-2)처럼 인접 리스트로 표현을 해보면 아래와 같이 작성한다.

A -> C -> E
B -> D
C -> B -> D -> E
D
E -> D

A -> C 5 -> E 2
B -> D 3
C -> B 6 -> D 9 -> E 8
D
E -> D 4

2-3) 자바의 입맛을 맞춘 인접 리스트 표현

우리가 정점을 알파벳으로 간단히 표현한다고 가정을 한다면 컬렉션에 있는 Map을 통해서 알파벳 정점, 문자열(정점의 인접 리스트) 표현을 하는 방법이 있다. 여기서 컬렉션에서 Map, Set의 개념이 무엇인지 잠깐 짚고 넘어가자.

-> Next Page ->

(1) Map, Set는 무엇인가요?
Collection 내부에는 List, Set, Queue, Stack, Map, Vector, Iterator 등등이 있다. 물론 C++에는 표준 STL이 있는데 우리는 자바를 이용해 공부를 하니 잠깐 Map이랑 Set에 대해서 잠깐 공부를 하고 넘어가 보자.
Map : Map은 영어로 지도라는 뜻이다. 하지만 List를 통해서 정수형 인덱스를 통해 검색을 시도했지만, Map을 이용하면 사용자가 원하는 키워드를 통해서 값을 얻어내는 방법(흔히 Key, Value를 주로 쓴다.)이라고 생각하면 쉽다. 더더욱 쉽게 이야기하면 정수형 인덱스뿐만 아니라 문자가 됐든 문자열이 됐든, 여러 객체로 검색을 한다는 뜻이다. 흔히 페이스북에 있는 #태그를 많이 봤는데 이와 관련된 타임라인이 나온다. 이를 접목시킨 개념으로 보면 쉽게 느껴질 것이다.
Set : 우리가 좋아하는 햄버거 세트, 추석이나 설날에 받는 스팸과 참치 세트의 개념이라고 생각하겠지만 실제로 수학 이론에서 많이 다루는 집합을 연상하면 되겠다. Set 내부에는 중복된 데이터가 저장이 될 수 없으며, 교집합, 차집합, 합집합 등의 기능도 포함이 되어있는데 실제로 우리는 포함됐는지 확인하는 contains 기능만 쓸 예정이다.

(2) Map으로 그래프 표현하기
여기서 우리가 아까 그래프 1를 참고해서 Map을 통해서 자바의 입맛에 맞춘 인접 리스트를 만들어 보면 아래와 같이 작성할 수 있다.

```
public static Map<Character, String>
createGraph(){
    Map<Character, String> graph=
    new HashMap<Character, String>();
    graph.put('A', "CE");
    graph.put('B', "CD");
    graph.put('C', "ABDE");
    graph.put('D', "BCE");
    graph.put('E', "ACD");
    return graph;
}
```

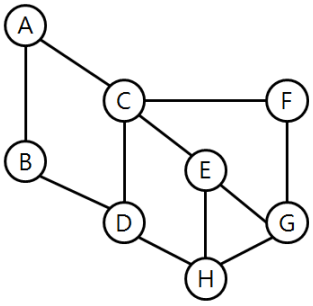
여기서 각 정점 별로 문자열을 통해서 인접한 정점들에 대해 오름차순으로 적어뒀다. 문자열로 작성을 하였더라도 크게 걱정할 필요가 없다. String에 있는 toCharArray()를 통해서 각 정점 별로 방문 여부를 확인하는 방법이 있기 때문이다. 이제 궁극적으로 공부해야 할 것이 코앞에 다가오고 있다는 의미로 받아들이면 좋겠다.

3. BFS(Breadth-1st Search)

그래프를 실제로 탐색하는 방법에는 깊이를 우선으로 하는 Depth-1st Search(DFS) 방법, 너비를 우선으로 하는 Breadth-1st Search(BFS) 방법 2가지로 나뉜다. 우리가 그래프 입문하기 요약정리에서는 여기까지 다루고 다음에는 최소 신장 트리에 대해서 공부를 해볼 것이다. 우선 BFS에 대해서 간단하게 알아보도록 하자.

3-1) BFS는 무엇인가?
BFS는 우리말로 너비 우선 탐색으로 해석할 수 있다. 직사각형에서 너비란 의미는 가로 길이로 생각할 수 있다. 그렇지만 그래프에서 너비는 시작 정점에서 인접하는 정점들에 대해서 방문을 하고 난 후 그 정점들에 대해 인접하는 정점들을 방문하는 의미로 받아들일 수 있다. 쉽게 이해하는 방법으로 아래와 같은 그래프를 통해 살펴보도록 하자.

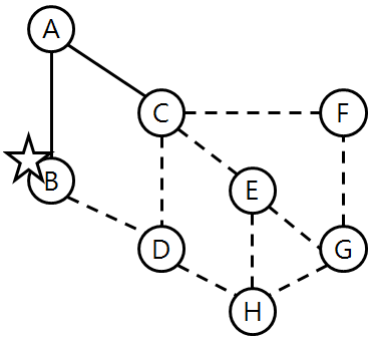
3-2) BFS의 과정은?



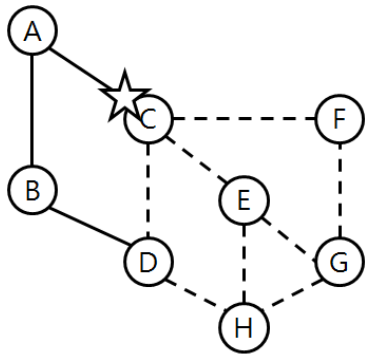
A	B, C
B	A, D
C	A, D, E, F
D	B, C, H
E	C, G, H
F	C, G
G	E, F, H
H	D, E, G

예를 들어 위와 같은 그래프가 있다고 가정을 하자. 그러면 이를 인접 리스트로(간단히 표로 그렸음) 나타내면 아래와 같이 나타낼 수 있다.
이를 이용하면 A를 이용해서 너비 우선 탐색을 해보자.

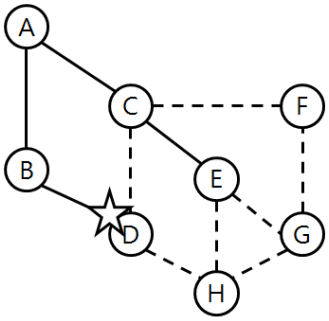
1> A를 방문하게 된다면
A와 인접하는 정점은 B, C가 되겠다. 그럼 B와 C를 방문을 하고 난 후에 방문 목록은 A, B, C가 되겠다. 여기서 다음 방문 목록은 B, C가 되겠다. 그럼 이 중에서 A의 인접 정점 중에서 최우선인 B로 간다.



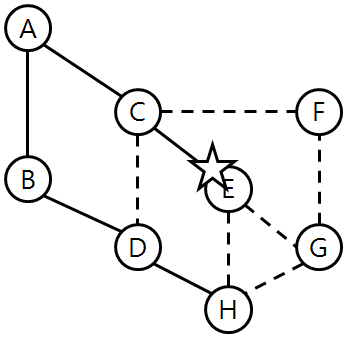
2> B를 재방문하게 된다면
B와 인접한 정점은 A랑 D이다. 그렇지만 A는 이미 방문을 했기 때문에 D를 방문해보도록 하자. 그럼 방문 목록은 A, B, C, D가 되겠다. 여기서 다음 방문 목록은 C, D가 되는데 다음 정점인 C를 족치러 가보자.



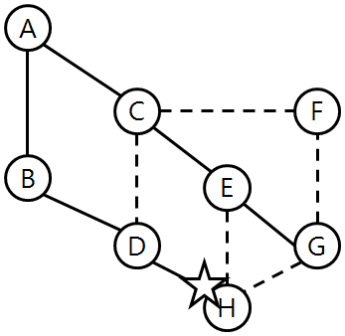
3> C를 재방문하게 된다면
C와 인접한 정점은 D랑 E이다. 그렇지만 D는 이미 방문을 했기 때문에 E를 방문해보도록 하자. 그럼 방문 목록은 A, B, C, D, E가 되겠다. 여기서 다음 방문 목록은 D, E가 되는데 다음 정점인 D를 살펴보러 가자.



4> D를 재방문하게 된다면
D와 인접한 정점은 C랑 H이다. 그렇지만 C는 이미 방문을 했기 때문에 H를 방문해보도록 하자. 그럼 방문 목록은 A, B, C, D, E, H가 되겠다. 여기서 다음 방문 목록은 E, H가 되는데 다음 정점인 E를 살펴보러 가자.

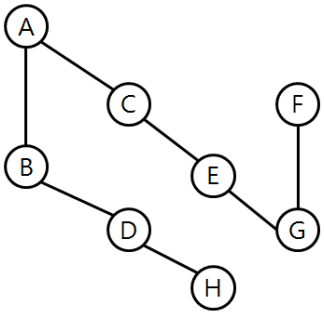


5> E를 재방문하게 된다면
E와 인접한 정점은 C랑 G이다. 그렇지만 C는 이미 방문을 했기 때문에 G를 방문해보도록 하자. 그럼 방문 목록은 A, B, C, D, E, H, G가 되겠다. 여기서 다음 방문 목록은 H, G가 되는데 다음 정점인 H를 족치러 가자.



6> H를 재방문하게 된다면
여기서 H의 인접한 정점은 D, E, G이다. 그렇지만 모두 방문을 했기 때문에 신장 트리를 만들어 가는 과정 중에 경로를 이어주면 순환이 생기기 때문에 신장 트리의 조건에 위배되어 다음 방문 목록인 G로 가보도록 하자.

7> G를 재방문하게 된다면
G의 인접한 정점은 E랑 F가 나온다. 그렇지만 여태껏 F는 한 번도 안 가봤기 때문에 F를 방문하는 것을 최종으로 너비 우선 최소 신장 트리가 완성되었다. 최종적인 신장 트리는 오른쪽 위처럼 그릴 수 있다. 그럼 BFS를 통해 탐색한 신장 트리가 나오게 되어 상황이 종료된다.



BFS의 목표대로 인접한 정점을 모두 방문하고 난 후 방문한 정점의 인접한 정점을 만나러 가는 과정을 그려보면 위와 같은 과정을 지녀야 한다. 여기서 신장 트리에 대해서 잠깐 짚고 넘어가보도록 하자.

<참고> 신장 트리(Spannin' Tree) : 트리의 일부인데 그래프 내에 모든 정점을 포함을 하는 트리로 볼 수 있고, 모든 정점들이 연결되어 있고 순환(사이클)을 포함하면 안 되고, 정점의 개수가 n개이면 n-1의 간선으로 연결이 되어 있다. 너비 우선 탐색을 하든 깊이 우선 탐색을 하든 간 신장 트리가 완성되게 되어있다. 쉽게 이야기를 하면 옛날의 원시인들이 집에서 근처 냇가나 야산에서 사냥하러 갈 때 남기는 발자국과 같은 의미로 생각해볼 수 있을까?

3-3) BFS 알고리즘 작성
BFS의 과정을 살펴보면 그렇게 어렵지는 않다. 이번에는 알고리즘을 천천히 살펴보면서 어떻게 돌아가는지에 대해 공부해보자.

```
public static void BFS(HashMap<Character,String>
그래프, char 시작정점) {
  HashSet<Character> 방문한정점 = new
HashSet<>(); // 1
  Queue<Character> 다음에방문할정점목록 = new
LinkedList<>();
// 2
  방문한정점.add(시작정점);
  다음에방문할정점목록.add(시작정점); // 3
  while (다음에방문할정점목록.isEmpty() == false){
    char 현재정점 = 다음에방문할정점목록.remove();
// 4
    System.out.printf("%c ", 현재정점); // 5
    String 인접정점목록 = 그래프.get(현재정점); // 6
    for (char 인접정점 : 인접정점목록.toCharArray()) // 7
    if (방문한정점.contains(인접정점) == false){
        방문한정점.add(인접정점);
      다음에방문할정점목록.add(인접정점); // 8
    }
  }
}
```

원래 자바에서는 한글을 잘 안 쓰는데 이해를 쉽게 하기 위해 교수님의 강의 노트를 참고해서 살펴보도록 하자.
1 : 이는 우리가 여태껏 방문했던 정점을 Set를 이용해 저장을 해주는 역할로 볼 수 있다. 아까 Set가 수학에서 쓰이는 집합의 원리로 쓰임을 살펴볼 수 있는데 방문한 정점 목록들이 중복으로 들어가지 않기 위해서 쓴 것으로 읽어볼 수 있다.
2 : 현재 정점에 있는 인접한 정점들을 방문한 목록을 그 때마다 적어 둬서 각 인접한 정점들의 또 다른 인접한 정점에 대하여 살펴보도록 하기 위해 일부러

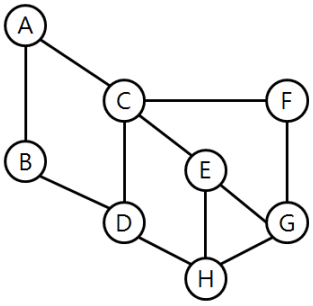
Queue를 이용해 작성하였다. Queue 자료구조는 아시다시피 FIFO(First-In-First-Out)의 원리를 적용해서 아까 과정의 2번부터 올바르게 돌아가게끔 하기 위해 이용했다고 보면 된다.
3 : 처음에 A라는 정점부터 시작을 하기 때문에 큐에 다음 정점 목록에도 A를 추가를 해뒀다. 이는 4번에 가면 A에 인접한 정점들을 가져와야 하기 때문이다.
4 : 현재 정점은 각 인접한 정점들을 저장시켜놓고 난 후에 기다리는 순서대로 나와서 또 다른 인접한 정점을 찾게끔 하여 BFS 탐색을 완료하는 역할을 한다.
5 : 이는 우리가 여태껏 방문한 정점들에 대해 최종 결과물로 출력을 해준다.
6 : Map을 이용하게 되면 정점 하나만 입력을 해서 그 정점에 인접한 목록(String으로 되어 있죠?)을 가져와서 후반에 for문을 통해서 각 인접한 정점에 대하여 방문을 하게끔 해준다. 2번 과정 중반을 살펴보면 인접한 B, C에 대해 방문을 하도록 리스트를 가져온다고 살펴보면 된다.
7 : 아까 필자가 그렇게 노래하던 부분이다. toCharArray() 메소드는 String에 있는 메소드의 일부로서 현재 문자열들을 char 형 배열로 바꿔주는 역할을 한다. 이의 요소들을 통해서 인접정점이란 변수를 통해 대입을 한다고 보면 된다. 여기서 : 의 역할은 고웹하신 분들이라면 쉽게 이해할 것이다.
8 : 방문한 정점 목록에 현재 정점의 인접한 정점을 넣어주고, 다음에 방문할 목록에도 써준다. 2번 과정에서 B, C를 일단 방문한 것으로 치고, 그 다음에 A에서 인접한 정점 중에서 오름차순 기준으로 먼저 방문하는 것으로 조건이 주어지면 다음방문할목록에 저장된 큐에서 B를 가져와서 B와 인접한 A, D의 정점을 통해서 3번 과정을 진행하면 되겠다.

각 실행 과정에 대해서 일부러 길게 쓴 이유는 여러분들이 쉽게 이해하기 위하여 작성하였다. 하지만 인접 리스트에 오름차순으로 저장을 권장하는 이유가 무엇일까? 적어도 ABC 순서로 출력이 되면 이해하기 쉽기 때문이다. 물론 인접 리스트는 내림차순으로 하든 무대포로 적어도 상관은 없지만 너비 우선 탐색에 대해 적어도 ABC 순서로 이해를 쉽게 하고 싶다면 오름차순으로 접근을 하길 권장하는 바이다.

4. DFS(Depth 1st Search)

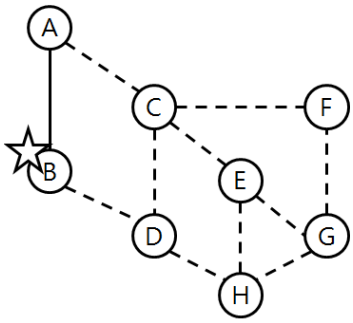
DFS를 우리말로 직역하면 깊이 우선 탐색으로 이해할 수 있다. 우리가 지하철역을 들어가 본다고 가정을 한다면 지상에는 버스 터미널이 있고, 지하 1층에 만남의 광장이 있고, 지하 2층에는 버스카드 찍는 곳, 버스카드 충전하는 곳, 꽃가게가 있고, 지하 3층에는 승강장이랑 자판기가 존재하기 나름이다.(3호선 남부터미널역을 기준으로 설명했어요...) 깊이의 기준은 무엇일까? 바로 정점의 인덱스 순이라고 말하고 싶다. 이번에는 DFS가 어떠한 과정으로 돌아가는지에 대해 알아보고 난 후 DFS 알고리즘을 공부해보도록 하자.

4-1) DFS의 과정은?
아까 BFS에서 다루었던 그래프를 기반으로 이번에는 DFS에 대해서 다루어 보도록 하자. 그래프는 오른쪽 위를 통해서 참고를 해보도록 하겠다.



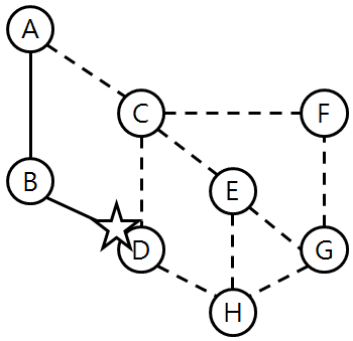
A	B, C
B	A, D
C	A, D, E, F
D	B, C, H
E	C, G, H
F	C, G
G	E, F, H
H	D, E, G

1> A를 방문하게 된다면



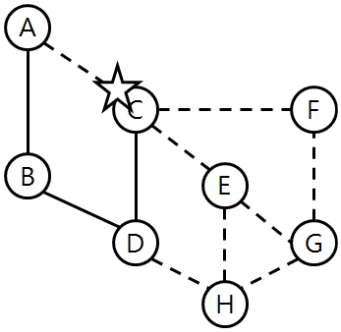
일단 A를 시발점으로 잡아보면 인접한 정점이 B랑 C가 있다. 여기서 깊이(알파벳 순서)가 가장 앞인 B를 선택해서 B의 인접한 정점들을 탐색해보자.

2> B를 방문하게 된다면



그러면 B에 인접한 정점 중에서 깊이가 가장 앞인 정점은 A가 되는데 A는 이미 방문을 한 상황이라서 D를 방문하는 수밖에 없다. 이제 D의 인접한 정점들을 탐색해보면...

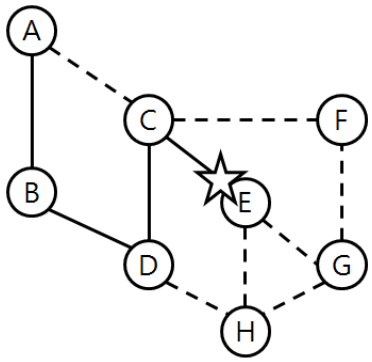
3> D를 방문하게 된다면



이제 D에 인접한 정점 중에서 깊이가 가장 앞인 정점은 B가 있는데 B를 방문하게 되면 신장 트리의 조건으로 간선이 2개일 수는 없는 판국이기 때문에 그 다음 최소인 C를 방문하는 수밖에 없다. 이제 C와 인접한 정점들을 족쳐보도록 하자.

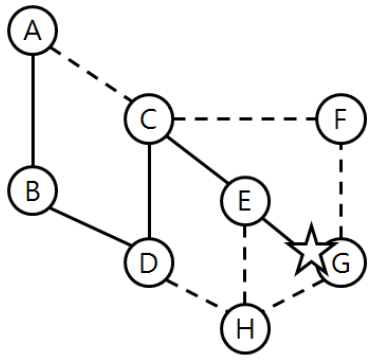
-> Next Page ->

4> C를 방문하게 된다면



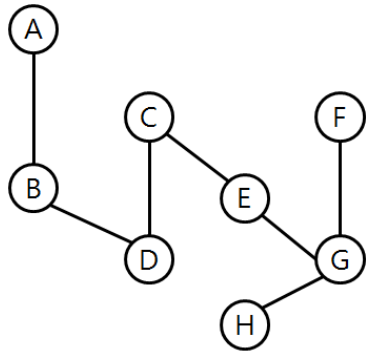
C의 인접한 정점들 중에서 깊이가 가장 앞인 정점은 A인데 A는 이미 방문한데다가 신장 트리의 조건으로서는 순환(사이클)이 없어야 하는 전제가 있기 때문에 A를 방문하지 못하고 그 다음으로 낮은 E가 방문을 하게 된다. 이제 E의 인접 정점들을 방문하게 된다면...

5> E를 방문하게 된다면



E에 인접한 정점은 G랑 H가 있다. 여기서 깊이가 가장 앞인 정점이 G가 있어서 G를 방문하면 되겠다. 이제 정점 G의 인접한 정점들을 탐색을 해보면...

6> G의 정점들을 방문하게 된다면...



G의 인접한 정점은 E랑 F랑 H가 있다. 일단 E는 이미 방문한 상태이기 때문에 F를 방문하게 되는데 여기서 F에 인접한 정점들은 모두 방문을 했기 때문에 H로 넘어가게 된다.(여기서 C랑 연결을 하면 신장 트리의 조건 중 순환이 있게 되어 전제 조건을 어기게 되어 안 된다.) 그럼 H에 인접한 정점들은 이미 다 방문을 하였기 때문에 최종적으로 DFS 신장 트리를 그리게 되면서 상황이 종료 된다. 방문 순서는 A, B, D, C, E, G, F, H가 되겠다.

깊이 우선 탐색은 깊이를 따지면서 신장 트리의 조건을 맞춰서 하면 되기 때문에 생각보다 쉽게 할 수 있을 것이다. 실질적으로 어떠한 알고리즘으로 돌아가는지에 대해 공부를 해보도록 하자.

4-2) DFS 알고리즘

```
static void DFS(HashMap<Character,String>
그래프, char 현재정점, HashSet<Character>
방문한정점) {
  방문한정점.add(현재정점);
  System.out.printf("%c ", 현재정점); // 1
  String 인접정점목록 = 그래프.get(현재정점); // 2
  for (char 인접정점 : 인접정점목록.toCharArray())
    if (방문한정점.contains(인접정점) == false){
      DFS(그래프, 인접정점, 방문한정점); // 3
    }
}
```

DFS 알고리즘은 이게 뭐미? 라고 할 정도로 재귀 호출로 짧게 구현할 수 있다. DFS를 탐색하는 원리는 바로 Stack을 이용했다고 볼 수 있지만, 여기서 재귀 호출은 Stack Segment를 사용했기 때문에 DFS 알고리즘이 적용된다. 그럼 이 알고리즘이 어떤 원리로 돌아가는지 한번 알아보자.

1 : 일단 현재 정점에서 방문한 기록을 맨 첫줄에 작성을 한다. 그럼 아까 그래프를 사례로 A를 먼저 방문했으니 A가 출력이 된다.

2 : 인접 정점 목록은 현재 정점에서 깊이가 가장 최소인 값들을 우선 방문을 하기 위해 가져오는 것으로 볼 수 있지만 여기서 깊이가 가장 최소인 뜻은 알파벳순으로 정점을 표현하면 먼저 오는 값을 골라내는 뜻으로 받아들일 수 있다.

3 : 인접한 정점들을 알파벳 순서대로 탐색을 해서 방문하지 않은 정점에 대해서 다음 차례에 방문 처리를 하도록 하는 역할을 한다. 아까 과정에서 A에 인접한 정점이 B, C가 될 수 있는데 알파벳 순(깊이가 얕은 순)으로 먼저 오는 B를 방문 처리를 하고, B의 인접 정점인 A, D 중에서 A는 방문했으니 D에 대해서 방문을 하는 원리로 생각을 하면 되겠다.

DFS 알고리즘을 재귀 호출을 통해 표현을 하는 방법이 있는데 실제로 자료 구조에서 Stack을 배운 적이 있을 것이다. Stack의 원리는 LIFO(Last in 1st Out)를 접목시켰다고 보면 되는데 쉽게 이야기하면 좀 짜증나겠지만 옛날에 학교에서 밥 먹는데 새치기를 한 친구를 생각해볼 수 있다. Stack의 원리를 접목시켜서 DFS의 원리를 공부해보자.

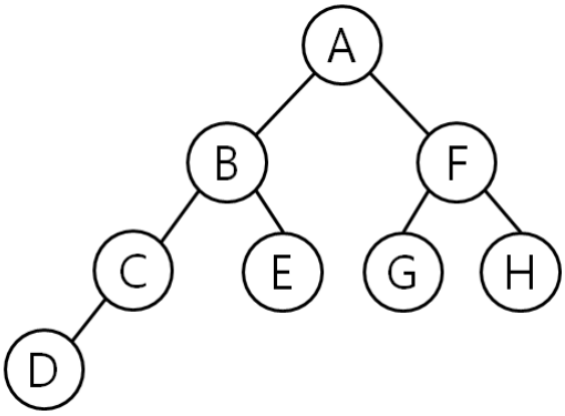
```
public static void DFS(HashMap<Character,String> 그래
프, char 시작정점) {
  HashSet<Character> 방문한정점 = new HashSet<>();
 Stack<Character> 다음에방문할정점목록 = new Stack<>();
  방문한정점.add(시작정점);
  다음에방문할정점목록.push(시작정점); // 1
  while (다음에방문할정점목록.isEmpty() == false) {
    char 현재정점 = 다음에방문할정점목록.pop(); // 2
    System.out.printf("%c ", 현재정점);
    String 인접정점목록 = 그래프.get(현재정점);
   for (char 인접정점 : 인접정점목록.toCharArray())
      if (방문한정점.contains(인접정점) == false) {
        방문한정점.add(인접정점);
        다음에방문할정점목록.push(인접정점);
      } // 3
  }
}
```

1 : Stack에 나오는 연산자는 push, pop으로 생각해 볼 수 있다. 다음에 방문할 목록에 대해서 Stack에 저장을 해두되 먼저 방문한 정점에 대해서는 맨 뒤에 저장이 된다고 보면 되겠다.
2 : pop 연산자는 현재 Stack에서 가장 나중에 방문한 정점에 대해 반환을 한다고 보면 되겠다. 가령 예를 들어 A와 인접한 정점들인 B랑 C를 방문처리를 하게 되면 분명 B가 나중에 저장이 되고, C가 먼저 나오게 된다. 그러면 아까 DFS 알고리즘에 대해서 위배될 수밖에 없다. 이런 경우에는 인접 리스트를 어떻게 바꿔야 할까? 인접 리스트의 순서를 뒤집는 방법이 있다. 여기서만 예외로 쓰인다고 알고 넘어가자.

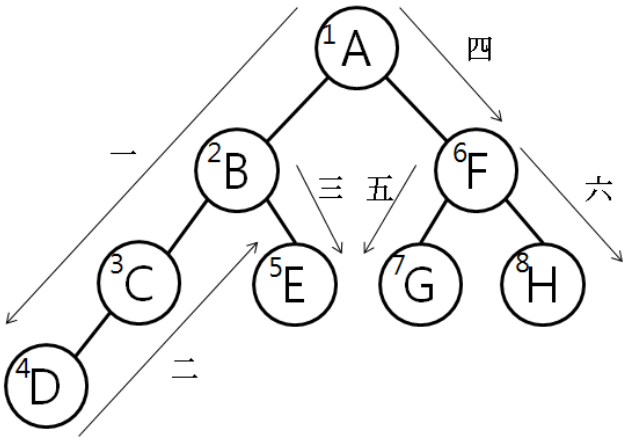
A	C, B
B	D, A
C	F, E, D, A
D	H, C, B
E	H, G, C
F	G, C
G	H, F, E
H	G, E, D

그러면 아까 정점 A랑 인접한 정점인 C를 방문 처리를 하고 난 후 B에 대해서 B랑 인접한 정점에 대해 방문한 기록을 적어두면서 발자취를 남기기 때문에 여기서는 인접 리스트를 뒤집어서 쓰는 것이 옳다고 생각할 수 있다.
3 : 각 인접 정점들에 대해서 방문 처리를 해주면서 Stack에 push 연산을 하는 모습을 볼 수 있다. 아까와 같은 인접 리스트를 push를 하게 된다면 애석하게도 알파벳 순서대로 느린 알파벳이 나오게 되니깐 이 이유를 토대로 2번처럼 뒤집어서 이용을 하는 방법을 고려해보도록 하자.

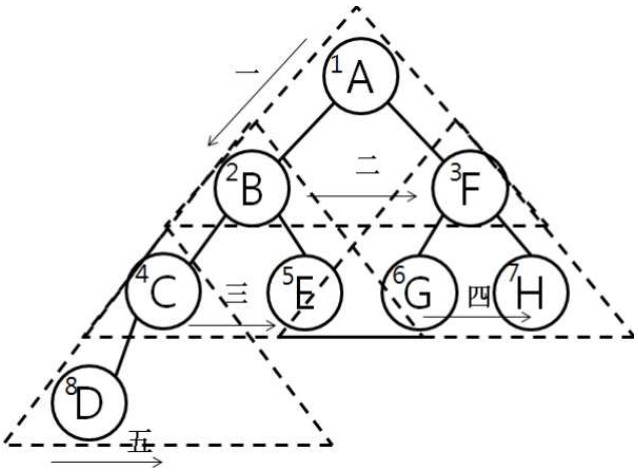
4-3) BFS와 DFS의 비기
우리가 그래프를 여태껏 이상한 모양으로 그려서 알아봤는데 이전에 공부했던 이진 트리(Binary Tree)를 잠깐 되새겨보도록 하자. 여기서 갑자기 이진 트리를 이야기를 하는지에 대해서는 BFS와 DFS의 원리를 쉽게 이해하기 위한 방법으로 설명하기 위해서 요약정리에 잠깐 적어뒀다.



물론 이진 트리도 그래프의 일종으로 받아들일 수 있다.(이진 트리⊂다진 트리⊂그래프) 그래서 인접 리스트에 대해서는 알파벳 순서대로 저장을 한다고 가정하고 넘어가자.



우선 A에서부터 깊이 우선 순회를 하게 되면 A, B, C, D까지 깊게 파다가 다시 B로 올라와 E를 방문하고, A로 올라와 이제 F를 방문해 G, H를 방문함으로서 끝난다. 순회 결과는 A, B, C, D, E, F, G, H가 되겠다. 우째 보면 이진트리의 순회가 갑자기 떠오를 것이다. 바로 이진트리의 전위 순회(Pre Order)의 원리가 깊이 우선 탐색의 원리와 비슷하기 때문이다. 이진트리를 깊이 순서대로 탐색을 하게 되면 결과는 깊이 우선 탐색과 같은 결과가 나오게 된다. 또한 이진 탐색 트리에서도 마찬가지로 작은 값은 왼쪽, 큰 값은 오른쪽을 탐색을 하게 되는데 이진 탐색 트리에 있는 값들을 탐색하는 방법도 깊이 우선 탐색의 원리를 접목시킨 원리로 생각하면 되겠다.



한편 이진트리를 BFS로 탐색을 해보면 어떤 결과가 나올까? 바로 A에서부터 시작해 B, F가 나오고, B의 인접한 정점들인 C, E가 나오고, F의 인접한 정점들인 G, H가 나오고 최종적으로 C의 인접한 정점인 D를 방문하게 된다. 그럼 최종적인 결과는 A, (B, F), (C, E), (G, H), (D)가 나오게 된다. 특별히 괄호를 친 이유가 이진트리의 같은 깊이(혹은 레벨) 별로 어떤 값들이 들어갔는지에 대한 결과가 나오게 되는 것이다. 이처럼 다른 그래프들을 BFS로 구현하면 깊이를 떠나서 자기 근처를 우선적으로 방문해서 발자취를 남기는 방법으로 구상을 할 수 있다. 깊이 우선 탐색과 너비 우선 탐색이 이해가지 않다면 이진트리를 통해서 어떠한 원리인지에 대해 감을 잡을 수 있는 기회를 가지도록 하자.